

L4: Practical loss-based stepsize adaptation for deep learning

Michal Rolínek & Georg Martius

Max-Planck-Institute for Intelligent Systems, Tübingen, Germany

{michal.rolinek,georg.martius}@tuebingen.mpg.de

February 19, 2018

Abstract

We propose a stepsize adaptation scheme for stochastic gradient descent. It operates directly with the loss function and rescales the gradient in order to make fixed predicted progress on the loss. We demonstrate its capabilities by strongly improving the performance of Adam and Momentum optimizers. The enhanced optimizers with default hyperparameters consistently outperform their constant stepsize counterparts, even the best ones, without a measurable increase in computational cost. The performance is validated on multiple architectures including ResNets and the Differential Neural Computer. A prototype implementation as a TensorFlow optimizer is released.

1 Introduction

Stochastic gradient methods are the driving force behind the recent boom of deep learning. As a result, the demand for practical efficiency as well as for theoretical understanding has never been stronger. Naturally, this has inspired a lot of research and has given rise to new and currently very popular optimization methods such as Adam [10], AdaGrad [5], or RMSProp [21], which serve as competitive alternatives to classical stochastic gradient descent (SGD).

However, the current situation still causes huge overhead in implementations. In order to extract the best performance, one is expected to choose the right optimizer, finely tune its hyperparameters (sometimes multiple), often also to handcraft a specific stepsize adaptation scheme, and finally combine this with a suitable regularization strategy. All of this, mostly based on intuition and experience.

If we put aside the regularization aspects, the holy grail for resolving the optimization issues would be a widely applicable automatic stepsize adaptation for stochastic gradients. This idea has been floating in the community for years and different strategies were proposed. One line of work casts the learning rate as another parameter one can train with a gradient descent (see [2], also for a survey). Another approach is to make use of (an approximation of) second order information (see [3] and [18] as examples). Also, an interesting Bayesian approach for probabilistic line search has been proposed in [14]. Finally, another related research branch is based on the “Learning to learn” paradigm [1].

Although some of the mentioned papers claim to “effectively remove the need for learning rate tuning”, this has not been observed in practice. Whether this is due to conservatism on the implementor’s side or due to lack of solid experimental evidence, we leave aside. Either way, in this paper we also attempt to change that.

Our strategy is performance oriented. Admittedly, this also means, that while our stepsize adaptation scheme makes sense intuitively, we do not provide or claim any theoretical guarantees. Instead,

we focus on strong reproducible performance against optimized baselines across multiple different architectures, on a minimum need for tuning, and on releasing a prototype implementation¹ that is easy to use in practice.

Our adaptation method is called **Linearized Loss-based optimal Learning-rate (L⁴)** and it has two main features. First, it operates directly with the (currently observed) value of the loss. This eventually allows for almost independent stepsize computation of consecutive updates and consequently enables very rapid learning rate changes. Second, we split the two roles a gradient vector typically has. It provides both a local linear approximation as well as an actual vector of the update step. We allow using a different gradient method for each of the two tasks.

The scheme itself is a meta-algorithm and can be combined with any stochastic gradient method. We report our results for the L⁴ adaptation of Adam and Momentum SGD.

2 Method

2.1 Motivation

The stochasticity poses a severe challenge for stepsize adaptation methods. Any changes in the learning rate based on one or a few noisy loss estimates are likely to be inaccurate. In a setting, where any overestimation of the learning rate can be very punishing, this leaves little maneuvering space.

The approach we take is different. We do not maintain any running value of the stepsize. Instead, at every iteration, we compute it anew with the intention to make maximum possible progress on the (linearized) loss. This is inspired by the classical iterative Newton’s method for finding roots of one-dimensional functions. At every step, this method computes the linearization of the function at the current point and proceeds to the root of this linearization. We use analogous updates to locate the root (minimum) of the loss function.

2.2 Algorithm

In the following section, we describe how the stepsize is chosen for a gradient update proposed by an underlying optimizer (e. g. SGD, Adam, momentum SGD). We begin with a simplified core version. Let $L(\theta)$ be the loss function (on current batch) depending on the parameters θ and let v be the update step provided by some standard optimizer, e. g. in case of SGD this would be $\nabla_{\theta}L$. Throughout the paper, the loss L will be considered to be non-negative.

For now, let us assume the minimum attainable loss is L^{\min} (we will come back to this in Section 2.4). We consider the stepsize η needed to reach L^{\min} (under idealized assumptions) by satisfying

$$L(\theta - \eta v) \stackrel{!}{=} L^{\min}. \quad (1)$$

We linearize L (around θ) and then, after denoting $g = \nabla_{\theta}L$, we aim for

$$L(\theta) - \eta g^{\top} v \stackrel{!}{=} L^{\min} \quad (2)$$

and thus the ideal stepsize is

$$\eta = \frac{L(\theta) - L^{\min}}{g^{\top} v}. \quad (3)$$

¹<https://github.com/martius-lab/l4-optimizer/>

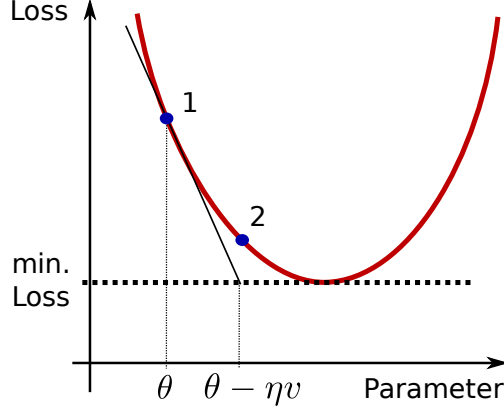


Figure 1: Illustration of stepsize calculation for one parameter. Given a minimum loss, the stepsize is such that the linearized loss would be minimal after one step. In practice a fraction of that stepsize is used, see Sec. 2.4.

First of all, note the clear separation between g , the estimator of the gradient of L and v , the proposed update step. Moreover, it is easily seen that the final update step ηv is independent of the magnitude of v . In other words, the adaptation method only takes into account the “direction” of the proposed update. This decomposition into the gradient estimate and the update direction is the core principle behind the method and is also vital for its performance.

The update rule is illustrated in Fig. 1 for a quadratic (or other convex) loss. Here, we see (deceptively) that the proposed stepsize is, in fact, still conservative. However, in the multidimensional case, the minimum will not necessarily lie on the line given by the gradient. That is why in real-world problems, this stepsize is far too aggressive and prone to divergence. In addition there are the following reasons to be more conservative: the problems in deep learning are (often strongly) non-convex, and actually minimizing the currently seen batch loss is very likely to not generalize to the whole dataset.

For these reasons, we introduce a hyperparameter α which captures the fixed fraction of the stepsize (3) we take at each step. Then the update rule becomes:

$$\Delta\theta = -\eta v = -\alpha \frac{L(\theta) - L^{\min}}{g^\top v} v, \quad (4)$$

Even though a few more hyperparameters will appear later as stability measures and regularizers, α is the main hyperparameter to consider. We observed in experiments that the relevant range is $\alpha \in (0.10, 0.30)$. Compared to, for example, SGD where the optimal learning rates vary over multiple orders of magnitude, this interval is very small. We chose the slightly conservative value $\alpha = 0.15$ as a default setting. We report its performance on all the tasks in Section 3.

2.3 Invariance to affine transforms of the loss

Here, we offer a partial explanation why the value of α stays in the same small relevant range even for very different problems and datasets. Interestingly, the new update equation (4) is invariant to affine loss transformations of the type:

$$L' = aL + b \quad (5)$$

with $a, b > 0$. Let us briefly verify this. The gradient of L' will be $g' = ag$ and we will assume that the underlying optimizer will offer the same update direction v in both cases (we have already established that its magnitude does not matter). Then we can simply write

$$\begin{aligned} -\alpha \frac{L'(\theta) - L^{\min'}}{g'^\top v'} v' &= -\alpha \frac{aL(\theta) + b - aL^{\min} - b}{ag^\top v} v \\ &= -\alpha \frac{L(\theta) - L^{\min}}{g^\top v} v \end{aligned} \quad (6)$$

and we see that the updates are the same in both cases. On top of being a good sanity check for any loss-based method, we additionally believe that it simplifies problem-to-problem adaptation (also in terms of hyperparameters).

It should be noted though that we lose this precise guarantee once we introduce some heuristical and regularization steps in Section 2.4.

2.4 Stability Measures and Heuristics

L^{\min} adaptation: We still owe an explanation of how L^{\min} is maintained during training. We base its value on the minimal loss seen so far. Naturally, some mini-batches will have a lower loss and will be used as a reference for the others. By itself, this comes with some disadvantages. In case of small variance across batches, this L^{\min} estimate would be very pessimistic. Also, the “new best” mini-batches would have zero stepsize.

Therefore, we introduce a factor γ which captures the fraction of the lowest seen loss that is still believed to be achievable. Similarly, to correct for possibly strong effects of a few outlier batches, we let L^{\min} slowly increase with a timescale τ . This reactivity of L^{\min} slightly shifts its interpretation from “globally minimum loss” to “minimum currently achievable loss”. This reflects on the fact that in practical settings, it is unrealistic to aim for the global minimum in each update. All in all, when a new value L of the loss comes, we set

$$L^{\min} \leftarrow \min(L^{\min}, L),$$

then we use γL^{\min} for the gradient update and apply the “forgetting”

$$L^{\min} \leftarrow (1 + 1/\tau)L^{\min}. \quad (7)$$

The value of L^{\min} gets initialized by a fixed fraction of the first seen loss L , that is $L^{\min} \leftarrow \gamma_0 L$. We set $\gamma = 0.9$, $\tau = 1000$, and $\gamma_0 = 0.75$ as default settings and we use these values in all our experiments. Even though, we can not exclude that tuning these values could lead to enhanced performance, we have not observed such effects and we do not feel the necessity to modify these values.

Numerical stability: Another unresolved issue is the division by an inner product in (4). Our solution to arising numerical instabilities are two-fold. First, we require compatibility of g and v in the sense that the angle between the vectors does not exceed 90° . In other words, we insist on $g^\top v \geq 0$. In our applications, this will be the case.

Second, we add a tiny ε as a regularizer to the denominator.

The final form of update rule then is:

$$\Delta\theta = -\alpha \frac{L(\theta) - \gamma L^{\min}}{g^\top v + \varepsilon} v, \quad (8)$$

with the default value $\varepsilon = 10^{-12}$.

2.5 Putting it together: L⁴Mom and L⁴Adam

The algorithm is called **Linearized Loss-based optimaL Learning-rate (L⁴)** and it works on top of provided gradient estimator (producing g) and an update direction algorithm (producing v), see Algorithm 1 for the pseudocode.

Algorithm 1 L⁴, a meta algorithm for stochastic optimization, compatible with momentum SGD, Adam or other gradient estimators, see Sec. 2. The default hyperparameters are: $\alpha = 0.15$, $\gamma = 0.9$, $\gamma_0 = 0.75$, $\tau = 1000$, and $\epsilon = 10^{-12}$.

Require: α : Stepsize/fraction
Require: γ, γ_0 : optimism loss improvement fraction
Require: τ : timescale of forgetting minimum loss
Require: $L(\theta)$: non-negative stochastic loss function w.r.t. parameters θ (a sample at step t is denoted by L_t)
Require: θ_0 : initial parameter vector
Require: $V(L, \theta)$: gradient direction function
Require: $G(L, \theta)$: gradient step function
 $t \leftarrow 0$
 $L^{\min} \leftarrow \gamma_0 \cdot L_0$ (fraction of initial loss)
while θ_t not converged **do**
 $t \leftarrow t + 1$
 $v \leftarrow V(L_t, \theta_{t-1})$ (gradient step)
 $g \leftarrow G(L_t, \theta_{t-1})$ (gradient estimator)
 $L_t^{\min} \leftarrow \min(L_{t-1}^{\min}, L_t)$ (minimum loss)
 $\theta_t = \theta_{t-1} - \alpha \cdot \frac{L_t - \gamma \cdot L_t^{\min}}{g^\top \cdot v + \epsilon} v$ (parameter update)
 $L_t^{\min} \leftarrow (1 + 1/\tau) \cdot L_t^{\min}$ (forgetting minimum loss)
end while
return: θ_t (final parameter vector)

For compactness of presentation, we introduce a notation for exponential moving averages as $\langle \cdot \rangle_\tau$ with timescale τ using bias correction just as in [10] (see Algorithm 2).

In this paper, we introduce two variants of L⁴ leading to two optimizers: (1) with momentum gradient descent, denoted by L⁴Mom, and (2) with Adam [10], denoted by L⁴Adam.

For L⁴Mom the update direction is given by

$$v = V_{Mom}(L, \theta) = \langle \nabla_\theta L(\theta) \rangle_{\tau_m}, \quad (9)$$

where $\tau_m = 10$ is the averaging timescale.

In case of L⁴Adam, the gradient step is given by:

$$v = V_{Adam}(L, \theta) = \frac{\langle \nabla_\theta L(\theta) \rangle_{\tau_m}}{\sqrt{\langle \|\nabla_\theta\|^2 L(\theta) \rangle_{\tau_s}}}, \quad (10)$$

with $\tau_m = 10$ and $\tau_s = 1000$ being the timescale for momentum and second moment averaging.

In both cases, the choice of $g = G(L, \theta) = V_{Mom}(L, \theta)$ ensures $g^\top v \geq 0$ as required in Section 2.4. Also, the averaged gradient is in practice often a more accurate estimator of the gradient on the global loss.

Algorithm 2 Bias corrected moving average

Require: τ : timescape
 $m_t \leftarrow 0$ (initialize mean with zero vector)
 $t \leftarrow 0$ (initialize step counter)
update_average(x): (x : input vector to be averaged)
 $t \leftarrow t + 1$
 $m_t \leftarrow (1 - 1/\tau) \cdot m_{t-1} + 1/\tau \cdot x$ (update average)
 return: $m_t / (1 - (1 - 1/\tau)^t)$ (correct bias)

3 Results

We evaluate the proposed method on four different setups, spanning over different architectures, datasets, and loss functions. We compare to the *de facto* standard methods: stochastic gradient descent (SGD), momentum SGD (Mom), and Adam [10].

For each of the methods, the performance is evaluated for the *best* setting of the stepsize/learning rate parameter (found via a fine grid search with multiple restarts). All other parameters are as follows: for momentum SGD we used a timescale of 10 steps ($\beta = 0.9$); for Adam: $\beta_1 = 0.9, \beta_2 = 0.999$, and $\varepsilon = 10^{-4}$. The (non-default) value of ε was selected in accordance with TensorFlow documentation to decrease the instability of Adam.

In all experiments, the performance of the standard methods heavily depends on the stepsize parameter. However, in case of the proposed method, the *default* setting showed remarkable consistency. Consistently across the experiments, already it outperforms the best constant learning rate for the respective gradient direction. In addition, the performance of our these default settings is also comparable with handcrafted optimization policies on more complicated architectures. We consider this to be the main strength of the L^4 method.

We present results for L^4 Mom and L^4 Adam, see Section 2.5. A prototype TensorFlow implementation of both optimizers will be made available during the review process. Throughout the experiments we have observed neither any runtime increase nor additional memory requirements arising from the adaptation.

As a general nomenclature, a method is marked with a * if the default parameters are used. In all other cases, the optimized learning rate is in place.

3.1 Solving badly conditioned regression

The first task we investigate is a linear regression with badly conditioned input/output relationship. It has recently been brought into the spotlight by Ali Rahimi in his NIPS 2017 talk, see [17], as an example of a problem “resistant” to standard stochastic gradient optimization methods. For our experiments, we used the corresponding code at [16].

The network has two weight matrices W_1, W_2 and the loss function is given by

$$L(W_1, W_2) = \mathbb{E}_{x \sim \mathcal{N}(0, I)} \|W_1 W_2 x - y\|^2 \quad (11)$$

$$y = Ax \quad (12)$$

where A is a badly conditioned matrix, i.e. $\kappa(A) = \sigma_{\max}/\sigma_{\min} \gg 1$, with σ_{\max} and σ_{\min} are the largest and the smallest singular values of A , respectively. Note that this in disguise a (realizable) matrix factorization problem: $L = \|W_1 W_2 - A\|_F^2$. Also, it is not a stochastic optimization problem but a deterministic one.

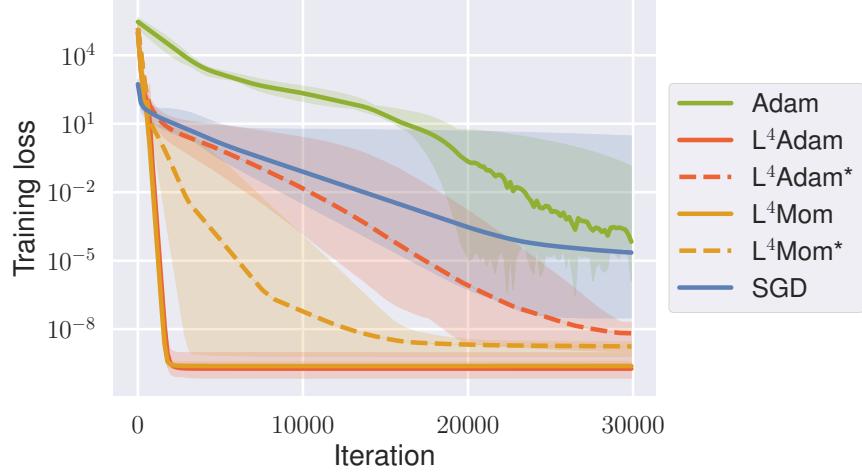


Figure 2: Training performance on badly conditioned regression task with $\kappa(A) = 10^{10}$. The mean (in log-space) training loss over 5 restarts is shown. The areas between minimal and maximal loss (after log-space smoothing) are shaded. For all algorithms the best stepsize was selected ($4 \cdot 10^{-5}$ and 10^{-3} for SGD and Adam respectively, and $\alpha = 0.25$ for both L^4 optimizers), except for the default setting marked with “*”. Note the logarithmic scale of the loss.

Figure 2 shows the results for $x \in \mathbb{R}^6$, $W_1 \in \mathbb{R}^{10 \times 6}$, $W_2 \in \mathbb{R}^{6 \times 6}$, $y \in \mathbb{R}^{10}$ (the default configuration of [16]) and condition number $k(A) = 10^{10}$. The statistics is given for 5 independent runs (with randomly generated matrices A) and a fixed dataset of 1000 samples.

We can confirm that standard optimizers indeed have great difficulty reaching convergence. Only a fine grid search discovered settings behaving reasonably well (divergence or too early plateaus are very common). The proposed stepsize adaptation method apparently overcomes this issue (see Fig. 2).

For comparison, we also include the classical Levenberg-Marquardt algorithm (LMA) [13] which can be viewed as a Gauss-Newton method with a trust region. In Tab. 1 the speed of the algorithms, both in terms of the number of iterations as well as wall-clock time² is reported. The same comparison is also performed on an instance of twice the size (all dimensions doubled).

The results show that the gradients provided by LMA reach convergence in a much smaller number of steps. However, at the same time, LMA is significantly more computationally expensive since each step involves solving a least squares problem. This can be clearly seen from comparing performance on the problem sizes in Tab. 1.

To summarize: L^4 with both Adam and Mom and default parameters can adapt to this difficult setting better than any constant learning rate on the badly conditioned regression task. With optimal parameters, it is converging very efficiently, much faster (in wall-clock time) than LMA.

3.2 MNIST Digit Recognition

The second task is a classical multilayer neural network trained for digit recognition using the MNIST [12] dataset.

We use the standard architecture with two layers containing 300 and 100 hidden units and ReLu activations functions followed by a logistic regression output layer for the 10 digit classes. The loss

²The experiments were conducted on a machine with i7-7800X CPU @ 3.50GHz with 8 cores.

Method	Steps	Time (s)
96 trainable weights		
L ⁴ Adam	2325 ± 765	1.4 ± 0.5
L ⁴ Mom	1606 ± 32	1.0 ± 0.02
LMA	68 ± 3	3.0 ± 1.4
192 trainable weights		
L ⁴ Adam	2222 ± 311	2.1 ± 0.4
L ⁴ Mom	1933 ± 116	1.8 ± 0.2
LMA	212 ± 114	123 ± 64

Table 1: Comparison with Levenberg-Marquardt algorithm. Time and the number of gradient updates needed to reach convergence ($L < 10^{-8}$) is reported. The average is with respect to 5 restarts. Two problem setups are considered, the default from [16] and its “scaled up by two” version. Stepsize $\alpha = 0.3$ was selected for LMA as the best performing one, and $\alpha = 0.25$ is chosen for both L⁴ optimizers.

Method	Test accuracy in %		
	1 epoch	10 epochs	30 epochs
Adam	95.7 ± 0.3	97.9 ± 0.3	98.0 ± 0.4
mSGD	96.4 ± 0.5	98.0 ± 0.1	98.5 ± 0.1
L ⁴ Adam	95.9 ± 0.7	98.4 ± 0.1	98.4 ± 0.1
L ⁴ Adam*	96.8 ± 0.2	98.3 ± 0.0	98.3 ± 0.1
L ⁴ Mom	95.4 ± 0.5	98.3 ± 0.1	98.4 ± 0.1
L ⁴ Mom*	96.3 ± 0.4	98.3 ± 0.1	98.4 ± 0.1

Table 2: Test accuracy after a certain number of epochs of (unregularized) MNIST training. The results are reported over 5 restarts.

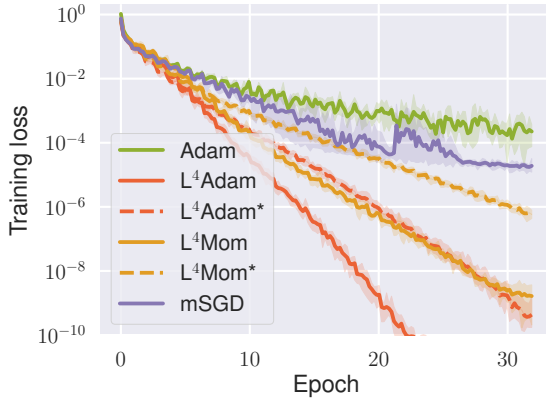
is given by cross-entropy. All weights are initialized in Xavier style [6] and batch size is 64.

Figure 3 shows the learning curves and the effective learning rates. The effective learning rate is given by η in (4). Note how after 22 epochs the effective learning of L⁴Adam becomes very small and actually becomes 0 around 30 epochs. This is simply because by then the loss is 0 (within machine precision) on every batch and thus $\eta = 0$; a global optimum was found. The very high learning rates that precede can be attributed to a “plateau” character of the obtained minimum. The gradients are so small in magnitude that very high stepsize is necessary to make any progress. This is, perhaps, unexpected since in optimization theory convergence is typically linked to decrease in the learning rate, rather than increase.

Generally, we see that the effective learning rate shows very nontrivial behavior. We can observe sharp increases as well as sharp decreases. Also, even in short time period it fully spans 2 or more orders of magnitude as highlighted by the shaded area. None of this causes instabilities in the training itself.

Even though the ability to generalize and compatibility with various regularization methods are not our main focus in this work, we still report in Tab. 3.2 the development of test accuracy during the training. We see that the test performance of all optimizers is comparable. This does not come as a surprise as the used architecture has no regularization. Also, it can be seen that the L⁴ optimizers reach near-final accuracies a bit faster after around 10 epochs.

(a) learning curve for 2-hidden layer NN on MNIST



(b) effective learning rate (MNIST)

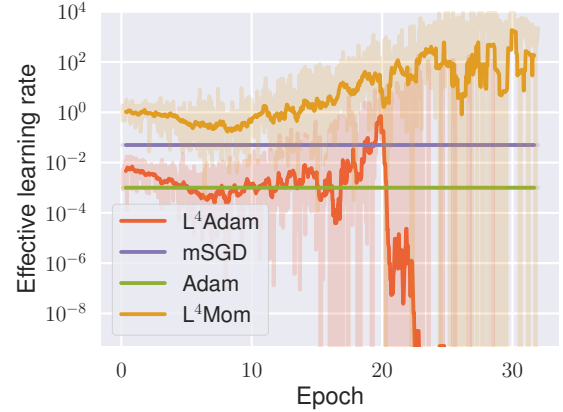


Figure 3: Training progress of multilayer neural networks on MNIST, see Section 3.2 for details. (a) Average (in log-space) training loss with respect to five restarts with a shaded area between minimum and maximum loss (after log-space smoothing). (b) Effective learning rates η for a single run. The bold curves are averages taken in log-space.

Comparison to other work: A list of papers reporting improved performance over SGD benchmarks on MNIST is long (examples include [18, 14, 1, 2, 15]). Unfortunately, there are no widely recognized benchmarks to use for comparison. There is a lot of variety in how the baseline optimizer is chosen (often the default setting for SGD, not an optimized one) and in how many training steps are reported (often fewer than one epoch). In this situation, it is difficult to make any substantiated claims. However, we believe that previous work does not achieve such rapid convergence as can be seen in Fig. 3.

3.3 ResNets for CIFAR-10 Image Classification

In the last two tasks, we target finely tuned publicly available implementations of well-known architectures and compare their performance to our default setting.

We begin with the deep residual network architecture for CIFAR-10 [11] taken from the official TensorFlow repository [20]. Deep residual networks [9], or ResNets for short, provided the breakthrough idea of identity mappings in order to enable training of very deep convolutional neural networks. The provided architecture has 32 layers and uses batch normalization for batches of size 128. The loss is given by cross-entropy with L^2 regularization.

The deployed optimization policy is momentum gradient with manually crafted piece-wise constant stepsize adaptation. We simply replace it with default settings of L^4 Mom and L^4 Adam.

The first surprise comes when we look at Fig. 3.3, which compares the effective learning rates. Clearly, the adaptive learning rates are much more conservative in behavior compared to MNIST, possibly signaling for different nature of the datasets. Also the L^4 Mom learning rate fairly well matches the manually designed schedule (also for momentum gradient). This match disappears after 150 epochs but from a quick inspection of Fig. 5 we see that the last 100 epochs of training are not very eventful anyway.

The performance comparison (see Fig. 5) brings another surprise. Both L^4 optimizers perform almost indistinguishable and outperform the default policy in terms of loss minimization (more strongly at the beginning than at the end). However, in terms of test accuracy, our optimizers eventually perform worse. By careful inspection of Fig. 5, we see the decisive gain happens right

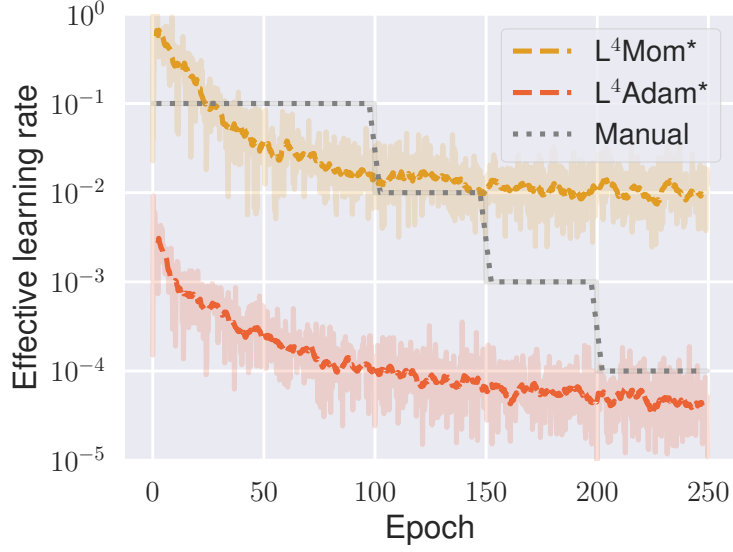


Figure 4: Effective learning rates η for CIFAR10. In grey, we see the provided learning rate decay schedule. The adaptive stepsize of L^4 Mom (which uses the same gradient type) is a decent fit in the first 150 epochs.

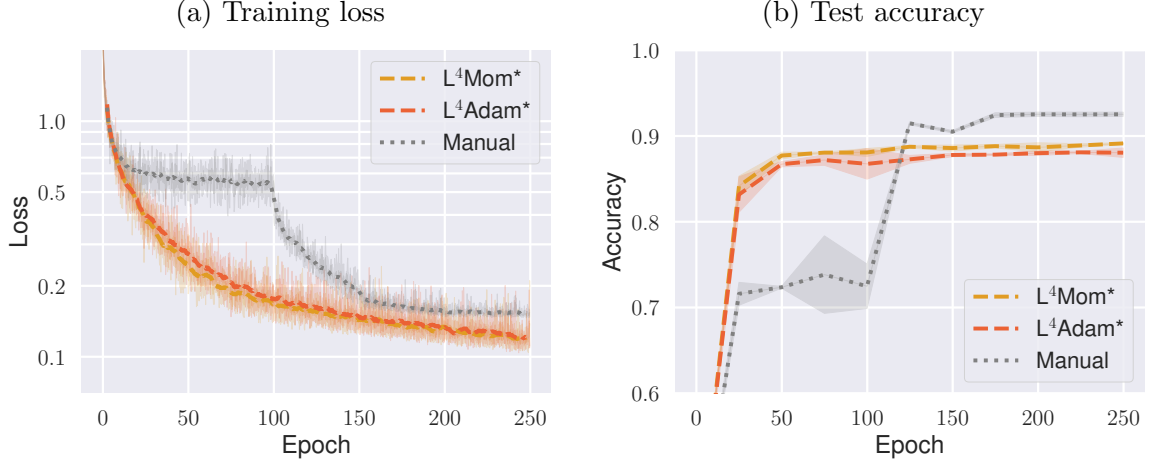


Figure 5: Training and test performance on ResNet architecture for CIFAR-10. Mean loss and accuracy are shown with respect to three restarts. Shaded regions cover the area between minimum and maximum. The default settings of the L^4 optimizers perform better in loss minimization, however, become inferior in test accuracy after the first drop in learning rate of the baseline's learning rate schedule (see also Fig. 3.3).

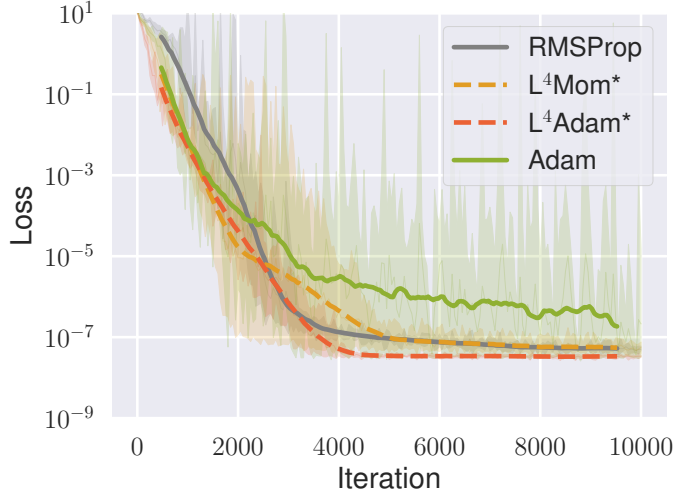


Figure 6: Training loss on the Differential Neural Computer architecture. Mean loss with min-max error bars (after log-space smoothing) with respect to five restarts is displayed. The L^4 optimizers use default settings, whereas RMSProp and Adam use best performing learning rates 0.005 and 0.01, respectively. We see high stochasticity in training, particularly with Adam. Both L^4 optimizers match RMSProp in performance.

after the first drop in the hardcoded learning rate. This, in itself, is very intriguing since both default and L^4 Mom use the same type of gradients of similar magnitudes. Also, it explains the original authors’ choice of a piece-wise constant learning rate schedule. To our knowledge, there is no satisfying answer to why piece-wise constant drops in learning rate lead to good generalization. Yet, practitioners use them frequently, perhaps precisely for this reason.

3.4 Differential Neural Computer

As the last task, we chose a somewhat exotic one; a recurrent architecture of Google Deepmind’s Differential Neural Computer (DNC) [8]. Again, we compare with the performance from the official repository [4]. The DNC is a culmination of a line of work developing LSTM-like architectures with a differentiable memory management, e.g. [7, 19], and is in itself very complex. The targeted tasks have typically very structured flavor (e.g. shortest path, question answering).

The task implemented in [4] is to learn a REPEAT-COPY algorithm. In a nutshell, the input specifies a sequence of bits a_n and a number of repeats k while the expected output is a sequence b_n consisting of k repeats of a_n . The loss function is a negative log-probability of outputting the correct sequence. The training input/output sequence pairs come in batches of size 16.

Since, the ground truth is a known algorithm, the training data can be generated on the fly, and there is no separate test regime. This time, the optimizer in place is RMSProp [21] with gradient clipping. We found out, however, that the constant learning rate 10^{-3} provided in [4] can be further tuned and we compare our results against the improved value 0.005. We also used the best performing constant learning rate 0.01 for Adam (with the suggested gradient clipping) as another baseline. The L^4 optimizers did not use gradient clipping.

Again, we can see in Fig. 6 that L^4 Adam and L^4 Mom performed almost the same on average, even though L^4 Mom was more prone to instabilities as can be seen from the volume of the orange-shaded regions. More importantly, they both performed better or on par with the optimized baselines.

We end this experimental section with a short discussion of Fig. 7, since it illustrates multiple

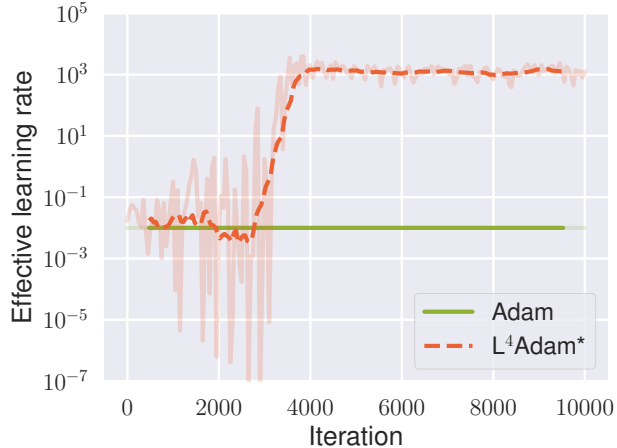


Figure 7: Effective learning rate η comparison of L^4 and plain Adam optimizers for the DNC architecture. The L^4 Adam displays a huge variance in the selected stepsize. This however has a stabilizing effect on the training progress (see also Fig. 6).

features of the adaptation all at once. In this figure, we compare the effective learning rates of L^4 and plain Adam. We immediately notice the dramatic evolution of the L^4 learning rate, jumping across multiple orders of magnitude, until finally settling around 10^3 . This behavior, however, results in a much more stable optimization process (see again Fig. 6), unlike in the case of plain Adam optimizer (note the volume of the green-shaded regions).

The intuitive explanation is two-fold. For one, the high gradients only need a small learning rate to make the expected progress. This lowers the danger of divergence and, in this sense, it plays the role of gradient clipping. And second, plateau regions with small gradients will force very high learning rates in order to leave them. This beneficial rapid adaptation is due to almost independent stepsize computation for every batch. Only L^{\min} and possibly (depending on the underlying gradient methods) some gradient history is reused. This is a fundamental difference to methods that at each step make a small update to the previous learning rate.

4 Discussion

We propose a stepsize adaptation scheme L^4 compatible with currently most prominent gradient methods. Two arising optimizers were tested on a multitude of datasets, spanning across different loss functions and network structures. The results validate the stepsize adaptation in itself, as the adaptive optimizers consistently outperform their non-adaptive counterparts, even when the adaptive optimizers use the default setting and the non-adaptive ones were finely tuned. The core design feature, ability to change stepsize dramatically from batch to batch was also validated. This default setting also performs well when compared to hand-tuned optimization policies from official repositories of modern high-performing architectures. Although we cannot give guarantees, this is a promising step towards practical “hyperparameter free” stochastic optimization.

We further suspect that the robustness of L^4 , along with its ability to actually drive loss to convergence create room for new regularization strategies, as well as for fresh analysis of the current ones. This could bring new insights into the connection between different types of regularization and generalization, possibly someday satisfyingly explaining events such as what was observed in Fig. 5 in the CIFAR-10 experiment.

References

- [1] Marcin Andrychowicz, Misha Denil, Sergio Gómez, Matthew W Hoffman, David Pfau, Tom Schaul, and Nando de Freitas. Learning to learn by gradient descent by gradient descent. In *Advances in Neural Information Processing Systems 29*, pages 3981–3989. Curran Associates, Inc., 2016.
- [2] Atilim Gunes Baydin, Robert Cornish, David Martinez Rubio, Mark Schmidt, and Frank D. Wood. Online learning rate adaptation with hypergradient descent. *CoRR*, abs/1703.04782, 2017.
- [3] R. H. Byrd, S. L. Hansen, Jorge Nocedal, and Y. Singer. A stochastic quasi-newton method for large-scale optimization. *SIAM Journal on Optimization*, 26(2):1008–1031, 1 2016.
- [4] Google Deepmind. Official implementation of the differential neural computer, 2017. <https://github.com/deepmind/dnc> Commit a4debae.
- [5] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, 12:2121–2159, July 2011.
- [6] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *AISTATS*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256. PMLR, 2010.
- [7] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines, 2014.
- [8] Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwiska, Sergio Gmez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, Adri Puigdomnech Badia, Karl Moritz Hermann, Yori Zwols, Georg Ostrovski, Adam Cain, Helen King, Christopher Summerfield, Phil Blunsom, Koray Kavukcuoglu, and Demis Hassabis. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476, October 2016.
- [9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, pages 770–778. IEEE Computer Society, 2016.
- [10] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *in Proceedings of ICLR*, 2015. arXiv preprint <https://arxiv.org/abs/1412.6980>.
- [11] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. CIFAR-10 (Canadian Institute for Advanced Research), 2009.
- [12] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998.
- [13] Kenneth Levenberg. A method for the solution of certain non-linear problems in least squares. *Quarterly of Applied Mathematics*, 2:164168, 1944.
- [14] M. Mahsereci and P. Hennig. Probabilistic line searches for stochastic optimization. In *Advances in Neural Information Processing Systems 28*, pages 181–189. Curran Associates, Inc., 2015.
- [15] Franziska Meier, Daniel Kappler, and Stefan Schaal. Online learning of a memory for learning rates. arXiv preprint <https://arxiv.org/abs/1709.06709>, 2017.

- [16] Benjamin Recht. Gradient descent doesn't find a local minimum, 2017. <https://github.com/benjamin-recht/shallow-linear-net> Commit d192d96.
- [17] Benjamin Recht and Ali Rahimi. Reflections on random kitchen sinks, 2017. <http://www.argmin.net/2017/12/05/kitchen-sinks>, Dec. 5 2017.
- [18] Tom Schaul, Sixin Zhang, and Yann LeCun. No more pesky learning rates. In Sanjoy Dasgupta and David McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28/3 of *Proceedings of Machine Learning Research*, pages 343–351, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR.
- [19] Sainbayar Sukhbaatar, Jason Weston, Rob Fergus, et al. End-to-end memory networks. In *Advances in neural information processing systems*, pages 2440–2448. Curran Associates, Inc., 2015.
- [20] TensorFlow GitHub Repository. Tensorflow implementation of ResNets, 2016. Commit 1f34fcf.
- [21] T. Tieleman and G. Hinton. Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning, 2012.